TinyCO – A Middleware Model for Heterogeneous Nodes in Wireless Sensor Networks

Atif Naseer
Science and Technology Unit
Umm Al-Qura University
Makkah, Kingdom of Saudi Arabia

Basem Y Alkazemi and Hossam I Aldoobi
College of Computer and Information System
Umm Al-Qura University
Makkah, Kingdom of Saudi Arabia

***Abstract*—Wireless sensor networks (WSNs) contain multiple nodes of the same configuration and type. The biggest challenge nowadays is to communicate with heterogeneous nodes of different WSNs. To communicate with distinct networks, an application requires generic middleware. This middleware should be able to translate the requests for contrary WSNs. Most of the wireless nodes use the TinyOS or Contiki operating systems. These operating systems vary in their architecture, configuration and programming model. An application cannot communicate with heterogeneous networks because of their divergent nature. In this paper, we design and implement TinyCO (a generic middleware model for WSNs), which overcomes these challenges. TinyCO is a general-purpose service-oriented middleware model. This middleware model can identify the heterogeneous networks based on TinyOS and Contiki. It allows applications to communicate with these networks using a generic request. This middleware interprets the given input into signatures of the underlying networks. This proposed middleware is implemented in Java and tested on TelosB motes.**

Keywords—wireless sensor networks; middleware; heterogeneous network; interoperability; service-oriented architecture

I. INTRODUCTION

Wireless sensor networks (WSNs) are composed of multiple sensor nodes with embedded sensors, actuators and radio communication [1,2]. These nodes are now capable of doing processing and transmission due to their efficient energy mechanism but are still complex in nature [3]. There was rapid development in applications of WSNs after the revolution of the Internet of Things (IoT). A recent survey from the Wireless World Research Forum foresees an increase in wireless devices up to 7 trillion by 2017 [4]. This would increase the demands of applications. An application cannot communicate directly with heterogeneous WSNs.

All the wireless nodes in any network contain an operating system to run. There are multiple operating systems available that serve WSNs. The two major operating systems used today are TinyOS and Contiki. To communicate with nodes, an application should know the signatures of the underlying network. All the applications are built for the specified network; these applications can only communicate with homogenous nodes of WSNs. All the nodes in WSNs use some operating system to communicate. This operating system remains the same in all sensor nodes within the same network and can vary in other WSNs. Most sensor networks use the TinyOS and Contiki operating systems. A WSN only consists of wireless nodes with the same type of operating system running.

In our previous paper [5], we identified the differences between the Contiki and TinyOS operating systems in their data exchange models. The operating systems are different in their architectures and programming models. That paper maps the architectures of TinyOS and Contiki into a component-based model. An application cannot communicate with both network types simultaneously without middleware. Middleware allows an application to communicate with heterogeneous nodes without modifying the request. The middleware will translate the request according to the signatures of the underlying network and will send the request to the network.

Atif et al. [6] proposed a general-purpose service-oriented middleware model for heterogeneous networks. Service-oriented architecture (SOA) is an advanced development in distributed computing. This approach uses "services" to interact with all the components of software and complete the task. All the frameworks following SOA have a common approach towards problems. Each activity in a service-oriented WSN application, like discovering, sensing and aggregation, is implemented as a separate service [7]. The proposed TinyCo middleware identifies the node types, configures the sensor nodes and allows data communication between heterogeneous networks. Initially the middleware will only support TinyOS- and Contiki-based WSNs. In this paper, we design and implement TinyCo (a generic middleware model for WSNs). Here, we discuss its complete working model, services, implementation and testing. TinyCo is implemented in Java and is tested on a real network of TinyOS and Contiki-based nodes. We deploy two different networks of TinyOS and Contiki of TelosB motes.

In the remainder of the paper, section II discusses the middleware role in WSNs and the challenges, section III highlights some of the common services of SOA-based middleware, section IV describes the TinyOS and Contiki programming models, the proposed middleware architecture is discussed in section V, section VI shows some implementation and results details, and section VII draws conclusions and offers suggestions for future work

II. RELATED WORK AND CHALLENGES

In WSNs, the middleware role is very significant in terms of data delivery and information retrieval. Nowadays, people

are building their own networks for multiple applications. An application cannot communicate with heterogeneous networks without middleware, as every network has potential mismatches in the data format and structures exchanged between nodes. An application layer of every network is responsible for data exchanges between users and the underlying network. Due to this mismatch, every network should have a separate application, and every application can only send/receive data to/from the specified underlying network. Middleware for WSNs plays an essential role in communication. The middleware identifies the network and modifies the request from an application according to the underlying network [8]. In the literature, many solutions have been discussed based on different approaches, like event-based, service-oriented, virtual-machine-based, agent-based, database-oriented and application-driven approaches. Several middleware models have been proposed under these approaches for WSNs.

A. Event-Based Middleware

Pietzuch [9] presented an event-based middleware model called Hermes. His proposed middleware supports reliability and interoperability between different components. The architecture of Hermes consists of a middleware layer, an event-based layer, a type-based and attribute-based pub/sub layer, an overlay routing network layer, and a network layer.

Boonma and Suzuki [10] proposed event-based middleware called TinyDDS. The architecture of this middleware allows an application to control the nonfunctional properties of the middleware and the application layer. This middleware model was designed specifically for a WSN and does not allow interoperability between heterogeneous nodes.

B. Service-Oriented Middleware

Some of the middleware follows a service-oriented approach. These types of middleware are based on SOA. SOA-based middleware is very common and well established in WSNs.

OASIS [11] is object-centric ambient-aware service-oriented sensor-net middleware. This middleware has a service-oriented programming framework. The major functionalities of this middleware are related to sensor node operation, communication and service discovery. Due to the limited resources of sensor nodes, this middleware maintains the service repository itself. There are two types of service repositories stored in this middleware: local and discovered.

Hydra [12] is middleware for ambient intelligence services and systems. Its architecture follows the component-based service-oriented approach. Some of the major components of its architecture are a service manager, an event manager, a device manager, a storage manager, a context manager and a security manager. These components provide services to its layers and allow an application to communicate with the underlying network.

C. Virtual-Machine-Based Middleware

Virtual-machine (VM) based middleware is also common in the literature. It provides a safe execution environment for user applications by virtualization. There are two types of VMs: middleware-level VMs and system-level VMs [13]. The middleware-level VMs are present between the application and the operating system, and system-level VMs are present inside the node. Every node in a network has a VM that allows applications to communicate with the network. These types of middleware consume more resources of nodes in terms of space and power.

Levis and Culler [14] proposed a VM-based middleware called Maté. The major contribution of Maté is to effectively handle resources like bandwidth and energy for sensor networks. Maté follows the event-based execution model of TinyOS. One of the main goals of this middleware is code management that provides updates to applications.

D. Agent-Based Middleware

The middleware that follows the agent-based approach is divided into modular programs. These programs are distributed through the network using mobile agents. Michal et al. [15] presented agent-based middleware for the IoT called Ubiware. This middleware supports the creation of multiple industrial systems. The major contribution of this middleware is to support the monitoring, composition, resource discovery and execution of multiple applications. This middleware is composed of three layers: a behavior engine, a middle layer, and shared and reusable resources (sensors, actuators, smart machines and devices).

UbiROAD [16] is agent-based middleware used for smart road environments. The major goal of this middleware is interoperability between in-car and roadside heterogeneous smart devices. This middleware provides a platform for smart traffic management. It can communicate with heterogeneous devices with respect to their standards, data formats and protocols. UbiROAD is self-adaptive middleware by deploying multiple agents.

E. Database-Oriented Middleware

The database-oriented middleware approach is very common nowadays. In this approach, an application can query a request to the database and the middleware executes that request. The sensor network receives the request from the middleware in the form of a query and sends the results accordingly.

Bonnet et al. [17] presented database-oriented middleware called COUGAR. This middleware deals with two types of data: stored data and data generated by sensor nodes. This middleware does not support event or code management but provides flexibility and accessibility to large groups of sensors

F. Application-Driven Middleware

The application-driven middleware approach focuses on quality of service and resource management. These types of middleware only support specific types of applications according to the network. These types of middleware fine-tune themselves according to the requirements of the application.

MiLAN [18] is application-driven middleware. It allows an application to send its requirements so that it can configure the network accordingly. To configure the network, MiLAN needs all the information of the network, like the number and types of

sensors. This middleware is mostly used in medical applications.

Alex et al. also designed application-driven middleware for TinyOS called TinyCubus [19]. This middleware framework is implemented on top of TinyOS and manages the requirements of an application. Some of the major applications are related to driver assistance systems and bridge monitoring.

G. Middleware Challenges

All types of middleware face certain challenges during their implementation and execution. Hadim and Mohamed [20] list the common middleware challenges for WSNs. Some of these challenges are:

- Limited resources
- Scalability
- Dynamic network topology
- Heterogeneity
- Dynamic network organization
- Data aggregation
- Quality of service
- Security

III. Services of SOA Middleware

SOA-based middleware is very popular nowadays due to its architecture. These types of middleware offer several services to applications and networks for the completion of tasks. Some of the commonly used services of such middleware are:

1) Node manager: This service manages the nodes and all corresponding services.

2) Service discovery: All the available services of the middleware are invoked by service discovery. Usually, the service ID is used to find out the service.

3) Data communication: This service is used to communicate data between the middleware and the network or application layer.

4) Network management: This service usually monitors the network performance. This service can also be used for network maintenance.

5) Notification: This service is used to notify about the events in the network.

6) Data gathering: This service gets the data from the network and makes it presentable for application.

7) Routing: The network routing protocols and algorithms are managed by this service.

8) Group management: Some of the middleware manages groups of nodes inside the network. This service allows the application to communicate with multiple groups.

IV. The TinyOS and Contiki Programming Models

Atif et al. [5] proposed a component-based model for the TinyOS and Contiki programming models. The operating systems are different in their architectures and programming models. "Component-based software engineering (CBSE) is a branch of software engineering that stresses the separation of concerns in respect of the wide-ranging functionality available throughout a given software system" [21]. Components communicate with each other through interfaces; these interfaces are also used for communication with other layers of the network. They map the programming models of TinyOS and Contiki into a component-based model. For that purpose, we use only some of the characteristics of CBSE, like initialization, state control, communication and data exchange.

Contiki has a modular architecture and follows the hybrid model. The Contiki architecture consists of the Contiki kernel, libraries, a program loader and processes [22]. These are like components. All Contiki programs are processes. A process act is a component and should have some core functionalities and some interfaces for interaction with other components.

TinyOS follows a component-based model using event-driven programming [23]. The TinyOS programming model supports a component-based approach and provides two types of components: modules and configurations. The interface of every component is implemented in module components, while configurations are used to assemble other components together, connecting the interfaces used by some components to the interfaces provided by others.

Figure 1 shows the component-based models of TinyOS and Contiki. To communicate across networks, the component-based approach supports data interoperability between heterogeneous networks.

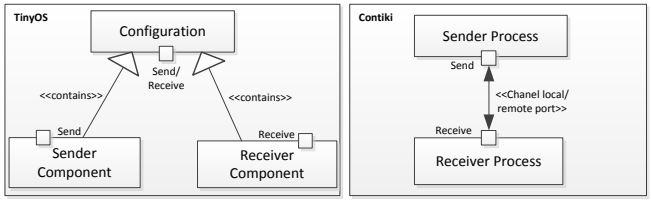

Fig. 1. Component-based models of TinyOS and Contiki

V. Proposed Middleware Model

A general-purpose middleware model was proposed in our previous paper [6]. The proposed middleware model consists of three layers: the application interface layer, the service layer and the hardware layer. These layers communicate the message from the application to the underlying network and vice versa. These layers provide services according to the needs of the network and the application. The user application interface (with the application layers of the middleware and the network) is interfaced with the hardware interface of the middleware. Figure 2 shows the layers of the proposed middleware.

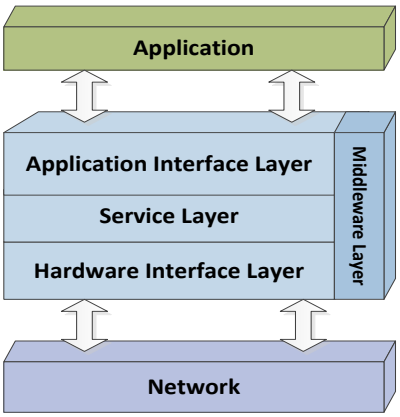

Fig. 2. Proposed middleware model

A. Application Interface Layer

The application interface layer is responsible for invoking services that bind the application and the middleware and allows the messages to be communicated between the application and the middleware. The message received by the application interface layer is passed to the service layer.

B. Service Layer

The service layer contains most of the services required for communication and discovery. This layer invokes its service after receiving the message from the application layer. The message header contains a request, which allows the middleware to decide which service should be invoked. This layer wraps the message according to the underlying network signature and passes the message to the hardware interface layer. Figure 3 shows the services of the middleware service layer. The major services of this layer are discovery, identification, configuration, routing and sensing.

1) Service discovery: This service is responsible for managing all the services of the middleware and identifies the appropriate service for the application according to its requirements. When a request comes from the application layer to the middleware layer, the service discovery calls the appropriate service from its stack and sends the request to that service.

2) Node identification: The node identification service identifies the type of nodes in the network. There are multiple base stations available through which an application interacts with the network. This service identifies the TinyOS and Contiki base stations and their port numbers.

3) Network configuration: The major responsibility of this service is to configure the network, its topology, its node types and its operating system.

4) Data sensing: The data sensing service collects the sensed data from the network and makes it presentable for application.

5) Routing: Network routing, protocols and algorithms are managed by the routing service of the proposed middleware. In our case, the routing for TinyOS and Contiki-based networks is managed through this service.

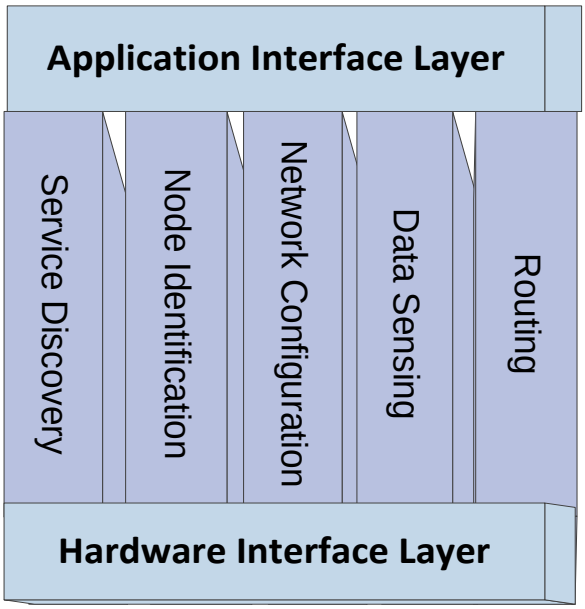

Fig. 3. Middleware layer services

C. Hardware Interface Layer

The hardware interface layer consists of open, close, read and write services. These services are invoked upon the arrival of the message from the service layer. The hardware interface layer is responsible for opening and closing the connection with the underlying network. This layer can read the message from a connected node and can send the message to the underlying network.

VI. Implementation and Results

To implement the proposed middleware model, we selected two widely used operating systems: TinyOS and Contiki. TinyOS and Contiki are used in many types of sensor nodes. We design two different WSNs based on TinyOS and Contiki and run our middleware to verify the results.

A. Sensor Nodes

A lot of sensor motes are currently used in the development of systems. In our experimentation, we are using MEMSIC's TelosB mote. The TelosB platform was developed by UC Berkley [24]. It is an open-source platform and has many feature:

- IEEE 802.15.4 RF transceiver
- 250 kbps data rate
- Onboard antenna
- 8 MHz microcontroller
- 1 MB external flash for
- data collection and programming
- Onboard light, humidity and temperature sensors
- Supports the TinyOS and Contiki operating systems

Fig. 4. TelosB mote

B. Operating Systems

The TelosB mote supports the TinyOS and Contiki operating systems. To prove the concept of the proposed middleware, we use both operating systems in different networks. They are different in their architectures and programming models. Our proposed middleware supports both types of network nodes with these operating systems.

TinyOS is used mostly in sensor nodes and it is the most robust, innovative, energy-efficient and widely used operating system. TinyOS was developed by the University of California [25]. It uses the NesC language for component implementation.

Contiki was developed by the Swedish Institute of Computer Sciences [26]. Contiki uses the C programming language; its application runs on a microcontroller. Contiki follows event-driven programming.

C. Model

To implement and test the proposed middleware, we design two different types of networks. The first network contains TinyOS nodes and the second network contains Contiki-based nodes. Both networks have two types of nodes: remote nodes and base stations. The base station is physically connected to the system and acts as a gateway between sensor nodes and the middleware. The middleware sends/receives all the messages to/from the network through the base station.

Figure 5 shows the implemented network model. This model contains an application, the middleware and multiple sensor networks. The application initiates any requests for the underlying sensor network through the middleware. The middleware translates the message according to the network signatures. The middleware is connected to the base stations of every network. The middleware transmits the message to the base station, which broadcasts the message in the network. Every network contains multiple remote sensor nodes. The major functionalities of these nodes are to sense the data and transmit it to the base station. In the scenario shown in Figure 5, there are two types of networks: one contains all the TinyOS nodes and the other contains all the Contiki nodes. These remote nodes are connected through radio link to the base stations and among themselves.

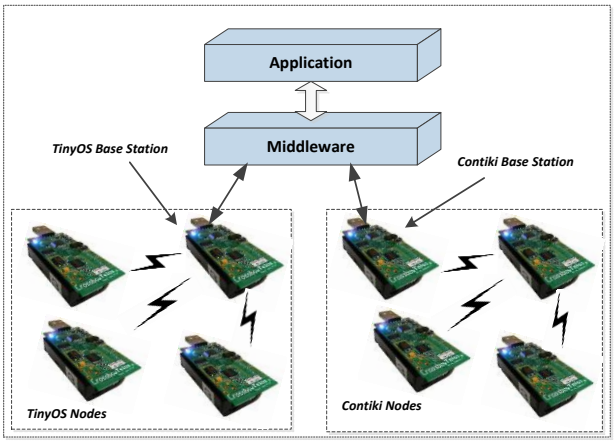

Fig. 5. Network model

D. Middleware Services

The proposed middleware allows applications to interact with different types of networks using a generic request. The major functionalities of this middleware are:

1) Identification of Connected Ports

The application requests the middleware to identify the ports connected. The middleware runs its service discovery service to find out the motes connected with any sensor node. In Figure 6, the complete process is explained by the flowchart. The service discovery service identifies all the connected ports. This service discovers the port number and displays it to the application.

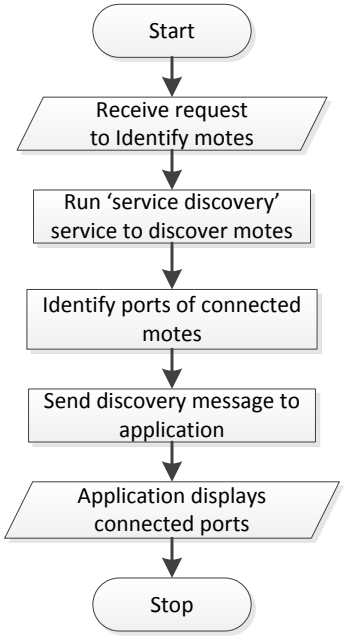

Fig. 6. Identification of connected ports

2) Identification of Base Station

After identifying the connected ports of the system, the application can identify the type of base station connected to the system. This identification is possible by invoking the network identification service of the middleware. This service sends the messages to all the connected nodes and identifies if it is a TinyOS or Contiki base station. The middleware receives the request from the application and converts it into two different message signatures: one for TinyOS and one for Contiki. The middleware sends these messages to every node connected to each port. The base station only receives the message as per its signature and rejects the other. The base station sends an acknowledgment to the middleware after receiving the message. Figure 7 shows the process of the network identification service in a flow chart. If the middleware finds some nodes other than TinyOS or Contiki, it also sends that information to the application. Once the middleware has received all the information of the connected nodes, it publishes the list to the application, along with the node type and the node ID.

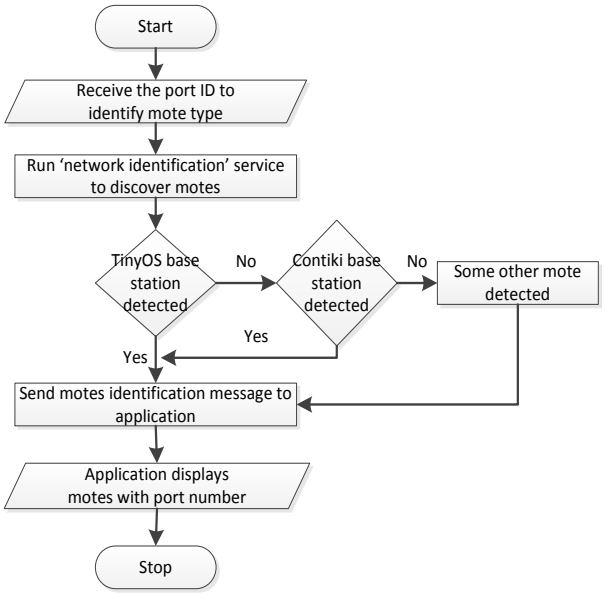

Fig. 7. Network identification

3) Identification of Remote Nodes

The middleware allows the application to identify the number of remote nodes in all the available connected networks. The application sends the request to the middleware to identify the number of nodes in the network. The middleware invokes its network configuration service to find out the total number of active nodes in the network. The middleware receives the message for some specific network, like TinyOS or Contiki. It converts the message as per the network message syntax and sends it to the specified network base station.

4) Data Communication

When the application needs to communicate with the underlying network, it requests data collection or sensing from the middleware. The middleware invokes its data sensing service, which manipulates the application request into specified network-identified signatures. The middleware sends the manipulated message to the base station of the TinyOS or Contiki network. The base station that broadcasts the message to the network and completes the request. The middleware also receives the sensed data from the underlying network through the base station and displays it to the application. Figure 8 shows the complete flow of data communication through the middleware.

E. Implementation

To implement the middleware, we first build the scenario as discussed in the above section. We develop two networks based on Contiki and Tiny OS nodes.

1) Contiki Network

The first network consists of Contiki nodes. There are two types of nodes: the base station and remote nodes. The base station is directly connected to the serial port and communicates with the middleware. To generalize the packet format, a new command packet is defined in the base station. Figure 9 shows the packet definition of the Contiki base station. This packet contains the following fields:

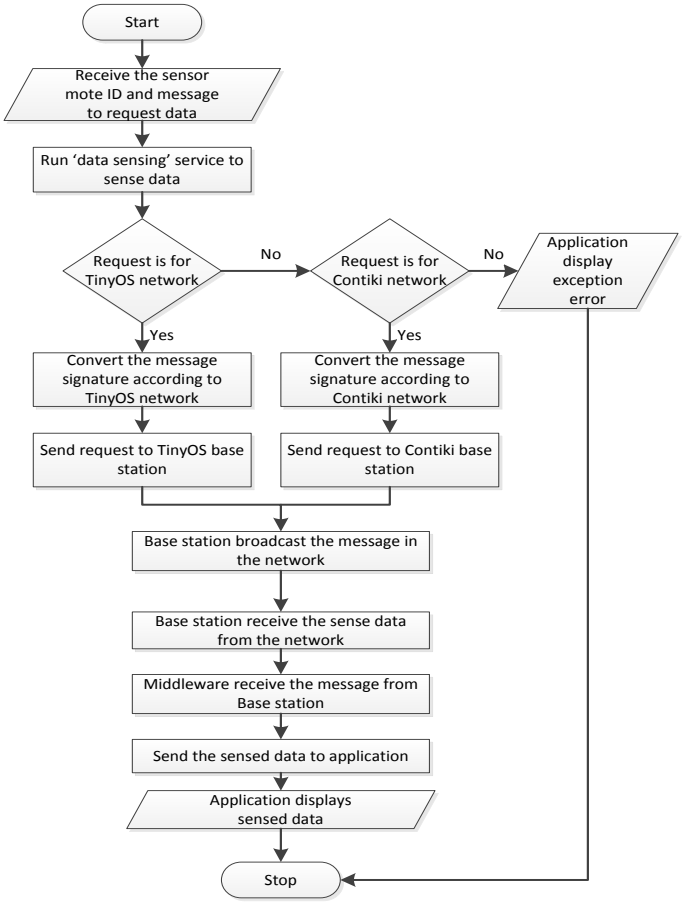

Fig. 8. Data communication

- **command:** used to store the command from the middleware
- **address:** used to store the address of the destination node

- **data:** used to store the data

```
struct command_packet {
  uint8_t cmd;
  uint8_t addr_0;
  uint8_t addr_1;
  uint32_t data;
};
```

Fig. 9. Contiki packet definition

As Contiki uses event-based programming, it waits for its serial line event to occur. Once this event has occurred, the middleware sends some command to the Contiki base station, which handles the following use cases:

a) ***Identify node type:*** The middleware sends a command to find out what type of base station is connected. Once the Contiki base station has received this command, it sends the message back to the middleware along with its node ID. After receiving this command, the base station will not broadcast the message into its network.

b) ***Data communication:*** The middleware sends this command with multiple variations. The base station is capable of fulfilling a certain number of requests from the middleware, like LEDs off/on, sensing the temperature and light, etc. Once the base station has received the command and destination address from the middleware, it sends this packet to the network. Every node of the network forwards this message to its neighbor node until it reaches the destination. The following forwarding method is used for packet forwarding within the network.

*static rimeaddr_t * forward (struct multihop_conn *c, const rimeaddr_t *originator, const rimeaddr_t *dest, const rimeaddr_t *prevhop, uint8_t hops)*

Every neighbor node of the base station receives the message using the receive method.

*static void recv (struct multihop_conn *c, const rimeaddr_t *sender, const rimeaddr_t *prevhop, uint8_t hops)*

Once the message has been received by the destination node, the node checks the command of the packet and completes the task. After sensing, the data is stored in the data part of the packet. The destination node sends this packet to the base station. Once the base station has received the packet from the remote node, it communicates with the middleware and sends the packet to the middleware for further action. Figure 10 shows the command execution for sensing temperature.

```
else if(cmd_pkt->cmd == CMD_4) //To Sense temperature
{
    leds_on(3);
    struct command_packet *data_pkt = malloc(sizeof(struct command_packet));
    data_pkt->cmd = CMD_0;
    data_pkt->data = get_temp();
    to.u8[0] = 1;
    to.u8[1] = 0;
    packetbuf_copyfrom(data_pkt, sizeof(struct command_packet));
    multihop_send(&multihop, &to);
}
```

Fig. 10. Middleware command execution

2) TinyOS Network

The second network in our scenario is based on TinyOS. All the nodes in this network are burned with TinyOS code. This network also contains a base station connected with the middleware and some remote nodes connected with the base station. The base station contains the base code of TinyOS, constituting two major files: BaseAppC.nc and BaseC.nc. TinyOS follows the component-based approach for programing the nodes. Hence, BaseAppC.nc contains the code for component declarations, and BaseC.nc contains all the implementations of the declared components. There are two types of interfaces used in the base station. The first one binds the base station to the middleware through serial communication, and the second one binds the base station to the remote nodes through radio communication.

The following methods are used in both types of interfaces.

a) ***SerialRequestSampleMsgsReceive:*** This method is used to receive the message from the middleware through the serial port.

b) ***RadioRequestSampleMsgsSend*****:** This method is used to send the request to the remote nodes through radio communication.

c) ***RadioSampleMsgReceive:*** This method receives the data from the remote nodes through radio communication.

d) ***SerialSampleMsgSend:*** This method is used to send the message back to the middleware through the serial port.

For remote nodes, we use SamplerAppC.nc and SamplerC.nc. SamplerAppC.nc contains the components and their bindings for remote nodes. SamplerC.nc contains the implementations of all these components. There are only two types of methods used for sending and receiving interfaces. Both interfaces communicate through radio. One interface is used to receive the message from the base station, and the second interface is used to send the data back to the base station. The following two methods are used in both types of interfaces.

a) ***RequestSampleReceive.receive:*** This method contains the code to receive the message from the base station or other neighbor nodes.

b) ***SampleSend.sendDone:*** This method contains the code to send the packet back to the base station. This packet contains the sensed data.

F. Results

The proposed middleware is implemented in Java. The middleware binds the application and underlying sensor networks with heterogeneous nodes. The initial version of this middleware facilitates the identification of motes, the discovery of their types and communication with heterogeneous networks. Figure 11 shows the proposed middleware design.

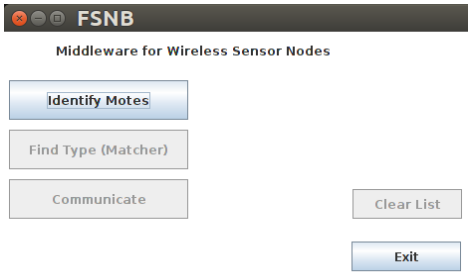

Fig. 11. Design of middleware model

1) Identification of Connected Ports

The first major functionality of the middleware is to identify the number of ports connected with base stations. The identify motes function identifies all such ports and displays the results. We use the mote interface functions of Java to identify the connected ports. Figure 12 shows the number of ports along with the mote types.

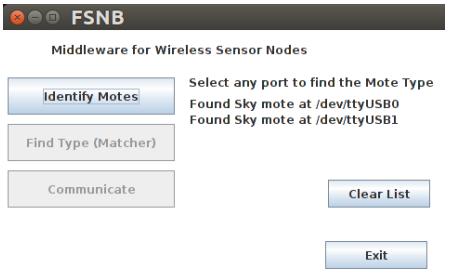

Fig. 12. Number of connected ports

2) Identification of Base Station

Once the mote has been identified, the next task is to identify the base station associated with every port. The matcher function of the middleware performs this task. This function takes a command as input and sends it to all the base stations connected with ports. Only those nodes that recognize the message with its signature receive the message. Once the node has received the message, it responds to the middleware about its type. Figure 13 shows that the base station connected with USB0 is a TinyOS node. Hence, the messages associated with TinyOS networks should be routed to this node through the USB0 serial port.

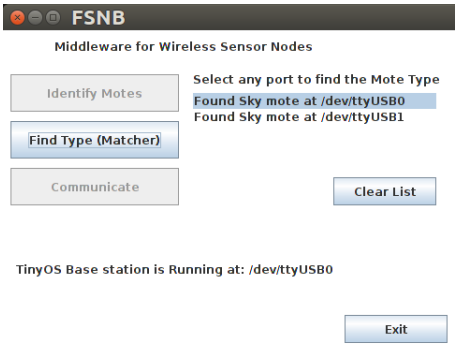

Fig. 13. Base station connection

The middleware records the ports and the base stations associated with them. Every time the matcher runs, it will identify all the nodes again. Figure 14 shows that a Contiki base station is running at USB1.

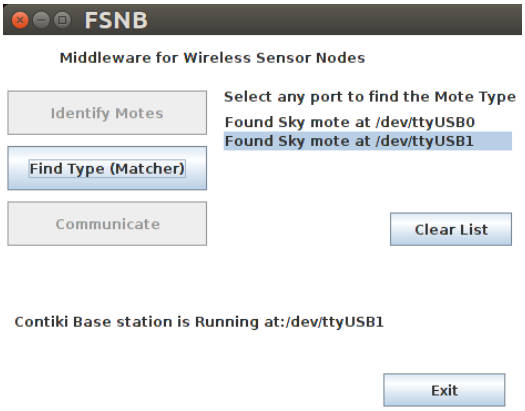

Fig. 14. Status of running base station

3) Data Communication

The major part of this middleware is to communicate between heterogeneous networks. The middleware receives the same message from applications for both types of networks; the middleware calls the respected API after identifying the network. Figure 15 shows the communication API for a Contiki network. It allows the user to set the LEDs and senses data like temperature and light. The middleware gets the remote node ID and action as an input and displays the sensed data to the middleware.

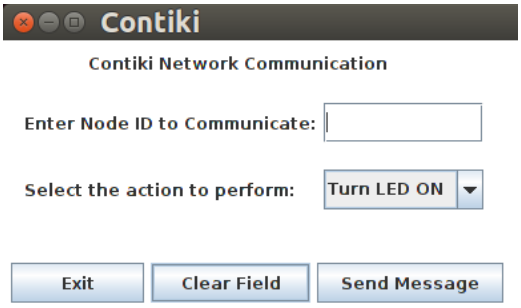

Fig. 15. Data communication API for Contiki

VII. Conclusion and Future Work

WSNs are composed of numerous sensor motes. These motes sense and transmit data. Some of the motes act as base stations to communicate with applications. Most sensor networks are composed of generic motes, and an application can communicate with these motes using their signatures. It is hard for an application to communicate with heterogeneous networks without middleware. Here, we implement general-purpose SOA-based middleware that lets an application communicate with two different types of networks with TinyOS and Contiki motes. We deploy a test bed to implement the proposed middleware and to run different scenarios to validate the results.

In future, we will enhance the functionality of this middleware for IoT applications. This middleware will be able to identify more sensor motes other than TinyOS and Contiki motes and will establish communication as well.

Acknowledgment

This work is funded by grant number 11-INF1674-10 from the Long-Term National Plan for Science, Technology and Innovation (LT-NPSTI), King Abdul-Aziz City for Science and Technology (KACST), the Kingdom of Saudi Arabia. We thank the Science and Technology Unit at Umm A-Qura University for its continued logistics support.

References

[1] K.L. Man, T. Krilavicius, Th. Vallee and H.L. Leung, "TEPAWSN: A formal analysis tool for wireless sensor networks," International Journal of Research and Reviews in Computer Science, Vol. 1, No. 1, pp. 24-26, 2010.

[2] Jo Ueyama, Danny Hughes, Ka Lok Man, Steven Guan, Nelson Matthys, Wouter Horré, Sam Michiels, Christophe Huygens and Wouter Joosen, "Applying a multi-paradigm approach to implementing wireless sensor network based river monitoring," Proc. ACIS International Symposium on Cryptography and Network Security, Data Mining and Knowledge Discovery, E-Commerce & Its Applications and Embedded Systems (CDEE), IEEE, 2010.

[3] K. L. Man, D. Hughes, S. U. Guan and P. W. H. Wong, "Middleware support for dynamic sensing applications," 2016 International Conference on Platform Technology and Service, Jeju, 2016.

[4] M. Uusitalo, "Global vision for the future wireless world from the WWRF," IEEE Veh. Technol. Mag., Vol. 1, No. 2, pp. 4-8, January 2006.

[5] A. Naseer, B. Y. Alkazemi and H. I. Aldoobi, "Component-based model for heterogeneous nodes in wireless sensor networks," Lecture Notes on Information Theory, Vol. 3, No. 1, pp. 25-30, June 2015. doi: 10.18178/lnit.3.1.25-30.

[6] A. Naseer, B. Y. Alkazemi and H. I. Aldoobi, "A general-purpose service-oriented middleware model for WSN," 2016 Eighth International Conference on Ubiquitous and Future Networks, Vienna, pp. 283-287, 2016.

[7] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema and J. Sztipanovits, "OASiS: A programming framework for service-oriented sensor networks," 2nd Intl. Conference on Communication Systems Software and Middleware, pp. 1- 8, 7-12 January 2007.

[8] Basem Y. Alkazemi, Atif Naseer and Emad A. Felemban, "Towards a general-purpose middleware model for WSNs: A literature survey," International Journal of Computer and Information Technology, Vol. 4, No. 1, January 2015.

[9] P. R. Pietzuch, "Hermes: A scalable event-based middleware," Univ. Cambridge, Comput. Lab., Tech. Rep. UCAM-CL-TR-590, http://www.cl.cam.ac.uk/techreports/UCAMCL-TR-590.pdf, Accessed October 2016.

[10] P. Boonma and J. Suzuki, "TinyDDS: An interoperable and configurable publish/subscribe middleware for wireless sensor networks," Principles and Applications of Distributed Event-Based Systems, 2010, p. 206.

[11] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema and J. Sztipanovits, "OASiS: A programming framework for service-oriented sensor networks", 2nd Intl. Conf. on Communication Systems Software and Middleware, pp. 1- 8, 7-12 January 2007.

[12] M. Eisenhauer, P. Rosengren and P. Antolin, "Hydra: A development platform for integrating wireless devices and sensors into ambient intelligence systems," *The Internet of Things*. New York: Springer, pp. 367-373, 2010.

[13] A. Azzara, S. Bocchino, P. Pagano, G. Pellerano and M. Petracca, "Middleware solutions in WSN: The IoT oriented approach in the ICSI project," in Proc. 21st Int. Conf. Softw. Telecommun. Comput. Netw., pp. 1-6, 2013.

[14] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks," SIGARCH Comput. Archit. News, Vol. 30, No. 5, October 2002.

[15] N. Michal, K. Artem, K. Oleksiy, N. Sergiy, S. Michal and T. Vagan, "Challenges of middleware for the Internet of Things," *Automation Control—Theory and Practice*. InTech, 2009.

[16] V. Terziyan, O. Kaykova and D. Zhovtobryukh, "Semantic middleware for context-aware smart road environments," Proc. 5th Int. Conf. Internet Web Appl. Serv., pp. 295-302, 2010.

[17] P. Bonnet, J. Gehrke and P. Seshadri, "Towards sensor database systems," in *Mobile Data Management*. New York: Springer, Vol. 1987, pp. 3-14, 2001.

[18] W. Heinzelman, A. Murphy, H. Carvalho and M. Perillo, "Middleware to support sensor network applications," *Network*, pp. 6-14, 2004.

[19] H. Alex, M. Kumar and B. Shirazi, "Midfusion: An adaptive middleware for information fusion in sensor network applications," Inf. Fusion, Vol. 9, No. 3, pp. 332-343, July 2008.

[20] S. Hadim and N. Mohamed, "Middleware: Middleware challenges and approaches for wireless sensor networks", IEEE Distributed Systems Online, Vol. 7, No. 3, 2006.

[21] Component-based software engineering, http://en.wikipedia.org/w/index.php?title=Componentbased_software_engineering, Accessed November 2016.

[22] Beginner's guide to crossbow motes, http://www.pages.drexel.edu/~kws23/tutorials/motes/motes.html, Accessed October 2016

[23] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh and E. Brewer, "TinyOS: An operating system for sensor networks", *Ambient Intelligence*. Berlin: Springer, pp. 115-148, 2005.

[24] TelosB data sheet, http://www.memsic.com/userfiles/files /Datasheets /WSN/telosb_datasheet.pdf, Accessed September 2016

[25] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler and K. S. J. Pister, "System architecture directions for networked sensors," SIGPLAN Not. 35 (11), pp. 93-104, ACM, 2000.

[26] TinyOS Platform hardware, http://docs.tinyos.net/tinywiki/index.php?title= Platform_Hardware&oldid=5648, Accessed September 2016.